Generation of Custom Co-processor Structure from C-Code

Jelena Trajkovic and Daniel Gajski

Technical Report CECS-08-05
June 27, 2008

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA

{jelenat,gajski}@cecs.uci.edu

Abstract

Designers of modern embedded systems may use two approaches for implementing an application: they can select an existing processor and map the application to it using a compiler, or they can design a custom processor for the code. While the secong approach provides optimality, the first one is cheaper and faster. To gain in terms of performance, cost and design time, we propose a method for extraction of application specific processor cores from its C code. Our approach consists of three phases. We start by quantifying the properties of the C code in terms of operation types, available parallelism and other metrics. We then create an initial data path to exploit the available parallelism. We then apply designer guided constraints to an interactive data path refinement algorithm that attempts to reduce the number of the most expensive components while meeting the constraints. Our experimental results show that our technique scales very well with the size of the C code. We demonstrate the efficiency of our technique on wide range of applications, from standard academic benchmarks to industrial size examples like the MP3 decoder. Each processor core was constructed and refined in under a minute, allowing the designer to explore several different configurations in much less time than needed for a manual design. We compared our selection algorithm to the manual selection in terms of cost/performance and showed that our optimization technique achieves better cost/performance trade-off. We also synthesized our designs with programmable controller and, on average, the refined core have only 23% latency overhead, twice as many block RAMs and 64% fewer slices compared to the respective manual designs.

Contents

1 Introduction **1**

2 Related Work **3**

3 Initial Data path Extraction **5**

4 Data path Optimization **6**
4.1 Resource Constraint Specification 6
4.2 Critical Code Extraction 7
4.3 Histogram Creation 8
4.4 Timing Overhead Computation 8
4.5 Balancing Instances of Unrestricted Components 10
4.6 Component Allocation and Net-list Creation 11

5 Results **11**
5.1 Interactive Design Exploration 12
5.2 Selection Algorithm Quality 18
5.3 Design Refinement Quality 20

6 Conclusions and Future Work **21**

7 Acknowledgments **22**

List of Figures

1 Comparison of design techniques 2
2 The data path is created first. It may be iterativly or manually improved. The controller is generated afterwords 3
3 Data path extraction technique 3
4 d_c computation for given resources constraints 7
5 ‘Spill’ Algorithm: computing the number of instances of multiplier for a given Timing Overhead 8
6 An Example of Extracted Data path Model. 10
7 Relative number of execution cycles on data paths generated for different architecture 13
8 Relative cycle time on data paths generated for different architecture 14
9 Relative execution time on data paths generated for different architecture 15
10 Total execution time on generated data paths 16
11 bdist2: Number of slices, number of RAMs and number of cycles for manually selected and automatically generated data paths 17
12 Sort: Number of slices, number of RAMs and number of cycles for manually selected and automatically generated data paths 18
13 dct32: Number of slices, number of RAMs and number of cycles for manually selected and automatically generated data paths 19
14 Mp3: Number of slices, number of RAMs and number of cycles for manually selected and automatically generated data paths 20
15 inv/forw 4x4: Number of slices, number of RAMs and number of cycles for manually selected and automatically generated data paths 21
16 Performance and area comparison for *dct32* 21

Generation of Custom Co-processor Structure from C-Code

Jelena Trajkovic and Daniel Gajski
Center for Embedded Computer Systems
University of California, Irvine

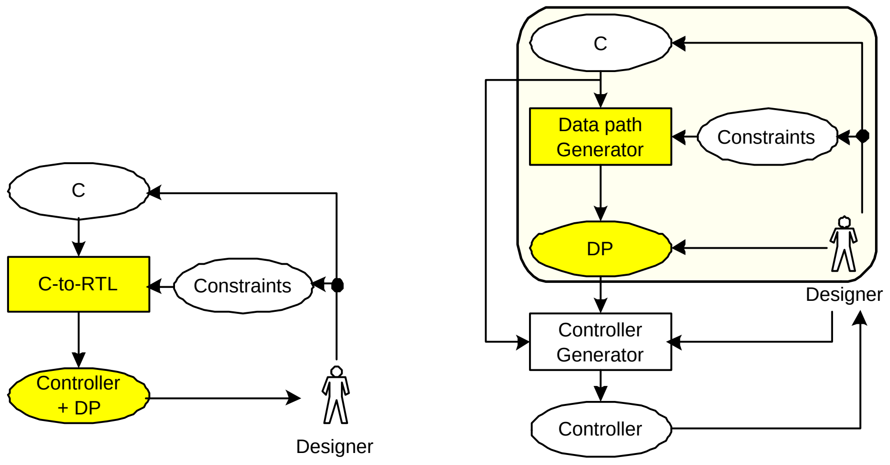
Abstract

Designers of modern embedded systems may use two approaches for implementing an application: they can select an existing processor and map the application to it using a compiler, or they can design a custom processor for the code. While the secong approach provides optimality, the first one is cheaper and faster. To gain in terms of performance, cost and design time, we propose a method for extraction of application specific processor cores from its C code. Our approach consists of three phases. We start by quantifying the properties of the C code in terms of operation types, available parallelism and other metrics. We then create an initial data path to exploit the available parallelism. We then apply designer guided constraints to an interactive data path refinement algorithm that attempts to reduce the number of the most expensive components while meeting the constraints. Our experimental results show that our technique scales very well with the size of the C code. We demonstrate the efficiency of our technique on wide range of applications, from standard academic benchmarks to industrial size examples like the MP3 decoder. Each processor core was constructed and refined in under a minute, allowing the designer to explore several different configurations in much less time than needed for a manual design. We compared our selection algorithm to the manual selection in terms of cost/performance and showed that our optimization technique achieves better cost/performance trade-off. We also synthesized our designs with programmable controller and, on average, the refined core have only 23% latency overhead, twice as many block RAMs and 64% fewer slices compared to the respective manual designs.

1 Introduction

In modern embedded system, a designer may map an application to a selected embedded processor or create a custom processor for the application. Using an existing processor saves time and money, but the mapping may be non-optimal. The existing processor may lack resources for efficient application execution, but have extra ones that remain unused (while consuming area and static power). On the other hand, custom, application specific pocessors include beneficial components and structures. Therfore, application specific processor cores are being increasingly used in modern embedded systems that demand high performance and low area/energy cost. In general, a processor core is defined by hardware components, structures and an instruction set, i.e. in hardware terms a data path and a controller. A design of such cores is non-trivial task. In most traditional C-to-RTL approaches (Fig. 1(a)), the decisions regarding components, structures and the instruction set are made at the same time. A problem of having many interdependent variables those tools solve by creating a structure and designing an instruction set as they ‘go through’ the code. Hence, optimizing the design is often very dificult. The problem is further exacerbated by the size and complexity of the application. Therefore, it is required that a design process is scalable. We overcome this problem by applying a design strategy where the data path and controller are designed separately. Our design technique is shown in Figure 1(b). The architecture is derived by analyzing the C code and designer specified constraints. The C code is then compiled into either control words for programmable cores or FSM for hardwired cores. Therefore, problem size reduction allows us to handle any size of C code.

Also, traditional C-to-RTL solutions have some numebr of constraints that may be used in order to guide the designe generation, but it is often hard or imposible either to pre-select a set of components or to change a part of design. As it may happen that the selected components, structures and instruction set need to be modified in order to satisfy design requirements, controlability is another crutial requirenment for the design process. In the proposed approach, designer specifies a set of resource constraints allowing him or her controlability over the component selection. In addition, after the data path has been



(a) Traditional C-to-RTL

(b) Proposed separate creation of data path and controller

Figure 1. Comparison of design techniques

automatically generated, the data path may be further fine tuned *manually*. Only when the data path has be finalized, the controller is generated.

Designers, typically, start with a C reference model of the application that needs to be implemented. The set of components, structures and connections is the most important factor that impacts design performance. They must be designed such that it can optimally execute the C reference. Thus, designers usually implement several alternatives (often in RTL-level) and evaluate them to fully explore the design possibilities. This is a tedious, error-prone process, that requires high level of expertise. We overcome this problem by using C as a starting point and iterativly optimizing the set of components and structure (Figure 1(b)). The designer can use our tool to try out several design alternatives in a short time. Our automatic architecture generation tool allows the designer to specify partial resource constraints. The remaining components of the architecture, their configuration and connection are derived automatically from the C code. For example, the designer may constrain the architecture to have two ALUs. Other architectural parameters, such as the size of register file and number of input/output ports, would be automatically derived. This is very helpful during the design process, because the designer does not need to think about multiple optimization variables, but can focus on the most important architectural parameters. The automation speeds up the generation and evaluation of different architectures.

Once the data path has been finalized, the C code and the generated data path are used to create a controller. The presented technique is flexible since it allows use of an arbitrary controller type with the generated data path. The controller may be fixed or programmable, may have instruction set and a decoder or a micro code, may run statically or dynamically scheduled operations and may be automatically or manually generated. This is a huge advantage of our approach over traditional C-to-RTL techniques that are limited by the number of states in the design.

Our target processor core template is shown in Figure 2. First, we construct the data path on the right by selecting, configuring and connecting various functional and storage units. For the constructed data path we develop the microcoded or hardwired controller. During core construction, the data path is automatically refined to meet the given performance and resource constraints. By separating data path and controller design, we allow simplified optimization of the controller by removing the data path optimization parameters from the equation. However, design of the data path architecture and controller that match an application C code are *two* problems. This paper deals with data path design only. The issues that arise from both control and data flow of the application were incorporated into our design technique as it will be described in Sections 4.3 and 4.4.

Figure 3 shows the steps of proposed extraction technique. In the first step, called *Initial Data path* (*IDp*) *Extraction*, source code for a given application is analyzed in order to extract code properties that will be mapped onto hardware components and structures and the *Initial Data path* is created. The Initial Data path is used for controller generation and profiling of given application code. The profiled code is then analyzed and the data path undergoes several iterations of refinement and estimation during the *Data path Optimization* step until the specified constraints have been satisfied.

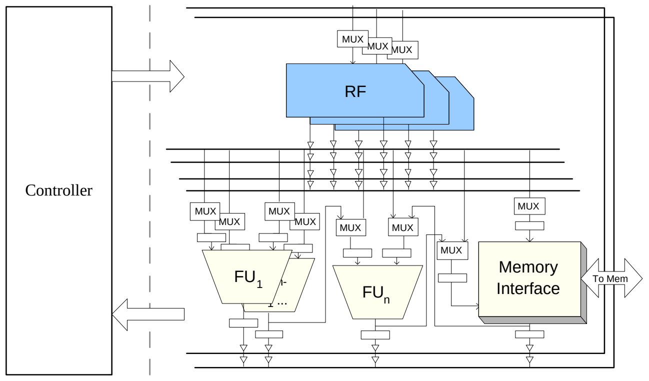


Figure 2. The data path is created first. It may be iterativly or manually improved. The controller is generated afterwords

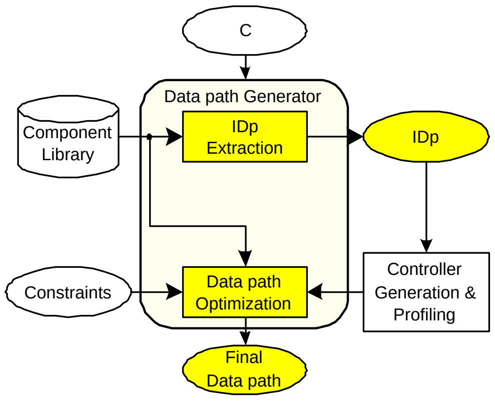


Figure 3. Data path extraction technique

As our experiments show, the generated designs have only 23% execution overhead compared to the manual design. This result was obtained with a data path architecture that *does not* implement custom forwarding, chaining, non-uniform pipelining or specialized functional units. We showed that even with *only basic* hardware components and templates implemented in the data path the quality and the design time are more than satisfactory.

Automating the design process in the proposed way brings several advantages. First, designers use their expertise to guide the tool instead of using it to do cumbersome and error prone HDL coding. Secondly, extraction and optimization algorithms generate data paths with much better cost/performance trade-off: relative to the manually extracted data paths from C, on average, number of slices is 50.08% less, number of RAMs is 39.94% less, while the average overhead in number of cycles is only 2.11%. Thirdly, the tool produces a working design in a fraction of time comparing to the manual process while having only 23% performance degradation, 2.29 times more BRAMs and only 64% of used slices. Moreover, the designer may change *only* constraints and repeat only optimization phase of the process instead of starting from the scratch. Finally, the generated design may be reused or fine tuned and only the controller would need to be changed. Also, the designer may explicitly control the type and quantity of components that would be included in the data path.

2 Related Work

In order to accomplish performance and power goals, ASIP/IS extension use configurable and extensible processors. One such processor is Xtensa [23], that allows the designer to configure features like memories, external buses, protocols and commonly used peripherals [1, 5]. Xtensa also allows the designer to specify a set of instruction set extensions, hardware for which is incorporated within the processor [8]. The extensions are formed from the existing instructions in style of VLIW, vector, fused operations or combination of those. Therefore, the customizations are possible only within bound of those combinations of existing instructions. This solution also requires the decoder modifications in order to incorporate new instructions. For example, having VLIW-style (parallel) instructions require multiple parallel decoders [8], which not only increase hardware cost (that may affect the cycle time), but also limits the possible number of instructions that may be executed in parallel. However, in our approach, the decoding stage has been removed. Therefore, there is no increase in hardware complexity and no limitations on number and type of operations to be executed in parallel. In case where the code size exceeds the size of on-chip memory, instruction caches and compression techniques may be employed, both of them have been in scope of our current research.

The IS extensions, in case of Stretch processor [22], are implemented using configurable Xtensa processor and Instruction Set Extension Fabric (ISEF). The designer is responsible for, using available tools, identifying the critical portion of the code ('hot spot') and re-writing the code so the 'hot spot' is isolated into the custom instruction. The custom instruction is then implemented in ISEF. Thus, the application code needs to be modified which requires expertise and potentially more functional testing. The designer is expected to explicitly allocate the extension registers. In contrary, our approach allows, but does not require C code modifications. Moreover, our approach does not require the designer to manipulate the underlying hardware directly.

On the other hand, C-to-RTL tools, such as Catapult Synthesis [13], Cynthesizer [6], and Behaviour Synthesizer Cyber [17] generate the data path and the controller simultaneously and to the best of our knowledge are applicable to fairly small code size. Catapult [13] allows control over resource sharing, loop unrolling and loop pipelining. It also porvides technology specific libraries [15] that allow specific hardware instances to be infered from C code. However, this requires code modifications. As reported in [14], code modifications took one week while the synthesis took one day. Also, the biggest listed C code had 480 lines of code. No other study in [16] reported on number of lines of C code. Behaviour Synthesizer Cyber [17], in addition to abovementioned, provides various knobs for fine tuning, such as multiple clocks, gated clocks, synchronous/asynchronous resert and synchronous/asynchronous/pipelined memory. The C code is extended to describe hardware by adding support for bit-length and in-out declarations; synchronozation, clocking and concurrency; and various data transfers (last two often not required). Such description is called behavioral C or BDL. Therefore, as seen in [27], the existing C code needs to be modified for in/out declaration, fifo requests, etc. In addition to control over loop unrolling and pipelining, Cynthesizer [6] also provides contol over operator balancing, array flattening, chaining, mapping of arrays or array indexes to memory, etc. The designer may also select a part of design to be implemented in gate level design in a given number of cycles.

In case of all of the tools, the data path is built 'on the fly' and heavily co-dependent on controller generation. Moreover, the resulting controler is usualy in FSM style. Use of the FSM imposes size constraints for the design. Some of the tools, like [6, 17], do provide FSM partitioning or hierarchical FSMs in order to expand beyond these constraints. Besides, while all do allow that a designer gives guideline to a tool, there is no mechanism by which a designer may influence a choice of particular components (other than inferring via code change in case of Catapult). Therefore, after the design has been made, designer may not make any modifications in the datapath. Contrary, the proposed technique separates creation of the data path and the controller, which automatically overcoms size constraint. Also, the designer may specify a subset of components and have the remainig of the data path automatically generated. Finally, the data path can be modified as little or as much after the automatic generation. Therefore, we find that providing designer with ability to control the automated design process and the ability to handle any size of C code are valuable assets in data path generation.

Many traditional HLS algorithms, such as [2, 18] create data path while performing scheduling and binding. [2] uses ILP formulation with emphasis on efficient use of library components, which makes it applicable to fairly small input code. [18] tries to balance distribution of operations over the allowed time in order to minimize resource requirement hence the algorithm makes decisions considering only local application requirements. [4] takes into account global application requirements to perform allocation and scheduling simultaneously using simulated annealing.In contrast with the previous approaches, we separate data path creation from the scheduling and/or binding i.e. controller creation. This separation allows: potentiall reuse of created data path by reprogramming, controllability over the design process, and use pre-layout information for data path architecture creation.

[11, 26, 25, 3, 12] separate allocation from binding and scheduling. [11] uses 'hill climbing' algorithm to optimize number and type of functional unit allocated, while [26] applies clique partitioning in order to minimize storage elements, units and interconnect. [25] use the schedule to determine the minimum required number of functional units, buses, register files and ROMs. Than, the interconnect of the resulting data path is optimized by exploring different binding options for data types, variables and operations. In [3] the expert system breaks down the global goals into local constraints (resource, control units, clock period) while iteratively moves toward satisfying the designer's specification. It creates and evaluates several intermediate designs using the schedule and estimated timing. However, all of before-mentioned traditional HLS techniques use FSM-style controller. Creation and synthesis of such state machine that corresponds to thousands of lines of C code, to the best of our knowledge, is not practically possible. In contrast to this, having programmable controller, allows us to apply our technique to (for all practical purposes) any size of C code, as it will be shown in Section 5.

Similarly to our approach, [12] does not have limitations on the input size, since it uses horizontally microcoded control unit. On the other hand, it requires specification in language other than C and it produces only non-pipelined designs, none of which is the restriction of the proposed technique.

Paper organization Our proposed technique is described in Fig. 3. It consists of two main steps: *Initial Data path Extraction* and *Data path Optimization*. Initial Data path Extraction (Section 3) extracts the properties of the input C code and maps them to the given components and component structures from the Component Library. Data path Optimization (Section 4.4) first select the portion of code to be optimized (Section 4.2), converts partial resource constraints to timing overhead (Section 4.1) and estimates execution characteristics of intermediate candidate designs (Sections 4.2 and 4.4) until the specified overhead is met.

3 Initial Data path Extraction

In order to extract the best possible data path architecture for a given C code, one needs to identify

- properties of the C code
- matching components for each property
- component structures/templates for given set of properties.

The set of properties of the C code include data types, operators, variables, parallelism, loops and dependencies. The components include functional units, storage elements (registers, register files, memories) and interconnect like buses, multiplexers and tri-state buffers, while structures/templates refer to different connectivity templates, like bus-based interconnect, multiplexer-based interconnect, dedicated connections, data path or component pipelining and load/store architecture model. For example, used data types and the number of used bits to represent them would determine the bit-width of used components; the available parallelism would affect selection of number of instances of functional units, number of registers or register file ports and pipelining.

The first step is to extract properties form the C code. Code property may be extracted from high level code, its CDFG or any other intermediate representation that front end of compile generates. We chose to start form an architecture independent schedule that assumes no resource constraints, such as ASAP and ALAP. We chose ALAP as the starting point since our experiments show that the parallelism is more evenly distributed across cycles than in corresponding ASAP. Following properties are extracted from code's ALAP schedule:

- OP: a set of operations for the given application code
- m_{op}: the maximum number of concurrent usages of operand op
- m_s and m_d: the maximum number of concurrent data transfers of the source and destination operands, respectively.

Furthermore, we define:

- $Ops(FU)$ as a set of operations that a functional unit FU performs
- $Selected$ set of selected functional units for the implementation of a final data path
- n_{FU} the number of instances of the functional unit FU in the $Selected$.

A matching heuristics $H(OP, Ops(FU)) \rightarrow Selected$ maps the set of operations OP to a set of functional units $Selected$ to implement the custom data path. The heuristics H determines both a type and the number of instances of a functional unit. Algorithm 1 describes the heuristics used in this work. $|\text{Ops}(FU_i)|$ represents the cardinal number of set Ops(FU_i). According to heuristics H, a functional unit FU_i will be selected to perform an operation op_i if it performs the greatest total number of operands alongside the chosen one. Therefore, this heuristics prioritizes functional units with a higher possibility for sharing. As for the number of instances, the heuristics includes an additional units of a chosen type to the set $Selected$

Algorithm 1 H(OP,Ops(FU))

```
for all $op_i \in OP$ do
    Select $FU_i$ such that
    $op_i \in Ops(FU_i)$ &&
    $|\text{Ops}(FU_i)| = \mathbf{max}(|Ops(FU_k)|, \forall k$ such that $op_i \in Ops(FU_k))$
    if $m_{op} > n_{FUi}$ then
        Add $[(m_{op} - n_{FUi}) \times FU_i]$ to $Selected$
    end if
end for
return $Selected$
```

only if the maximum number of occurrences of operand op_i is greater than the number of units n_{FU} of that type currently in *Selected*, i.e. if $m_{op} > n_{FUi}$. For example, if application requires 3 units to perform additions and 4 units to perform subtractions, and an ALU is chosen for both addition and subtraction operator, the tool will allocate only 4 instances of the ALU.

The derived required number of source operand transfers m_s determines the number of register file output ports and the number of source busses, where the number of destination operand transfers m_d translates into register file input ports and the destination buses. The register file ports are selected using heuristics described in [24].

To ensure that the interconnect is not a bottleneck in the Initial Data path, the connection resources are allocated in greedy manner. This means that output ports of all register files are connected to all source buses. Similarly, input ports of all register files are connected to all destination busses. The same connection scheme applies to the functional units and the memory interface, making it possible to transfer any source operand to any functional unit or memory interface and the result back into a register file. Note that data forwarding is not presented in this paper and that it is a part of our current research.

A hardware component in the Component Library is represented by a data structure that consists of the unique ID, component type, number and type of input and output ports, delay, area and power. A functional unit also has a list of operations that it performs.

4 Data path Optimization

Data path optimization uses information from two sources: schedule and execution profile. The schedule provides cycle-by-cycle usage of every resource within a basic block and the profile provides the execution frequency for each of the blocks. Therefore, we define a dynamic schedule as a schedule of an application where each basic block is annotated with its execution frequency. The Initial Data path is compiled and profiled in order to create the dynamic schedule. The dynamic schedule alongside with the constraints specified by the designer and the Component Library are input to the data path optimization step.

Our data path optimization method proceeds as follows. First, the basic blocks to be optimized are selected based on the criteria that will be shown in Section 4.2. Then, for each selected basic block a histogram (usage per cycle) for each data path resource is created. The histogram is also annotated with the frequency of a corresponding basic block. For each specified resource constraint, the estimation algorithm computes the Timing Overhead. After that, the algorithm traverses the constrained design space, creating new designs and estimating the overhead for each new design. If the current refinement step does not produce a design that performs within Timing Overhead, the algorithm backtracks to the previous design, marks the refinement effort as 'tried' and continues modifying the previous design. Once all designs have been explored, the output net-list is created. Following sections provide details of each step of the data path optimization.

4.1 Resource Constraint Specification

In practice, completely automatic data path selection has not been shown to be better or even at par with manual design. We address this problem by providing control to the designer to put constraints on the chosen components while creating the data path. In the one extreme case, the designer may completely specify the data path, thereby rendering the optimization trivial. Alternately, the designer may chose to specify no constraints. In that case, the optimization algorithm produces a best-effort result. The Resource Constraint (RC) specification is simply a set of bounds for the number of chosen components.

The constraints may be given partially. For example, constraint

$$1 \leq num(adder32) \leq 2 \quad (1)$$

implies that the data path must have at least one 32-bit adder but not more than two. If the designer wishes to let the tool decide the number of multipliers, no constraint may be specified for adders. The optimization algorithm ensures that components with specified lower bounds are not optimized away aggressively.

Each of given constraints is used to compute the number of extra cycles (d_c) that are required for application execution if the upper bound of instances of the component is used (Figure 4). To compute d_c, we use the estimation algorithm that will be shown in Section 4.4. Once d_c values for all constraint components have been computed, the smallest one is chosen to be the Timing Overhead. The Timing Overhead value is used to compute the number of instances for all unrestrained components (Section 4.5).

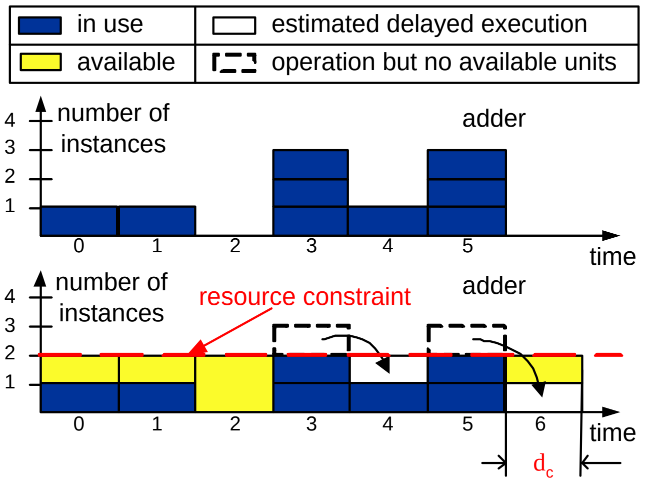


Figure 4. d_c computation for given resources constraints

4.2 Critical Code Extraction

The goal of this step is to select basic blocks in the source code that contribute the most to the execution time and have the largest potential for optimization. The question is how to decide which basic blocks are the most promising. Our selection criterion is based on the relative size and the relative execution frequencies of the basic blocks in the application. It is likely that very large basic blocks have high potential for optimization, since they have several operations that may potentially be performed in parallel. On the other hand, even minor optimization of basic blocks that have high frequency will yield high overall performance gains. Finally, we have a class of basic blocks that have average length and average frequency, so an average reduction in their length will yield overall performance improvement comparable to the improvement from previous two types of optimization.

Following our optimization policy, we keep 3 lists of pointers to the basic blocks. The first list is sorted by frequency-length product, the second by length and the third by frequency. We use the parameterizable metrics to decide if the block is to be included in the list of blocks for optimizations. For frequency-length product we use:

$$f_i \cdot l_i \geq P_{fl} \cdot \sum_{j=0}^{N} f_j \cdot l_j \quad (2)$$

where f_i and l_i are frequency and length (number of cycles) of the basic block i, mfl_i is frequency-length product, P_{fl} is parameter specified by the designer and N is the total number of blocks in the application. The block i is considered for optimizing if inequality 2 is satisfied.

In case of the list sorted by length, we observe the length of the block l_i and

$$l_i \geq P_l \cdot \max_{j=0}^{N}(l_j) \quad (3)$$

where P_l is length parameter specified by the designer. The current block is considered for optimizing if inequality 3 is satisfied.

Given f_i as the frequency of the basic block i,

$$f_i \geq P_f \cdot \max_{j=0}^{N}(f_j) \tag{4}$$

where P_f is frequency parameter. We include the block i in the optimization candidate list if inequality 4 is satisfied.

List creation is performed during histogram creation; hence it does not contribute significantly to the overhead in the execution time of the Data path Optimization. The lists contain only pointers to the basic blocks and do not introduce any space overhead. The optimization algorithm will be applied only to the selected subset of blocks.

4.3 Histogram Creation

A histogram shows cycle-by-cycle usage for each data path resource. A histogram is created for each selected basic block for:

- each component type, in case of functional units and buses
- for data ports of the same kind (input or output), in case of the storage units.

It is important to group the items of the same kind (i.e. all adders or all source buses) together in order to easily estimate potential impact on execution while changing the number of instances of a particular data path resource. Histograms are extracted from the schedule generated for the Initial Data path. The example of a histogram for adders is shown in the Figure 4. The shown basic block has 6 cycles (0 to 5). It can be seen that no instance of adder is used in cycle 2, one instance is used during cycles 0, 1 and 4, and three instances are used in the cycles 3 and 5. If we assume that the type and number of instances of all other components (other FUs, memories, register files and its ports, buses and multiplexers) do not change, we can conclude that we need 3 instances of adder to execute this basic block in no more than 6 cycles.

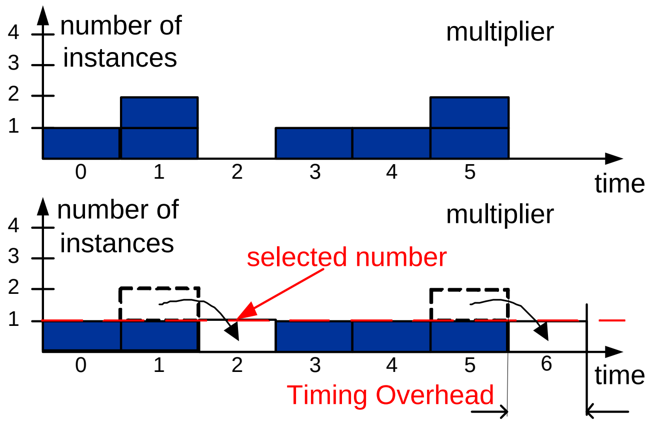


Figure 5. ‘Spill’ Algorithm: computing the number of instances of multiplier for a given Timing Overhead

4.4 Timing Overhead Computation

As described in Section 4.1 the designer specifies constraints in a form of lower and upper bound for the number of instances for resources in the data path. Changing the number of instances of a resource may affect the application execution time. As seen in the example in the Figure 4, if the number of adders is changed from three to two, we may expect that the number of cycles required for execution of the basic block would increase. The Algorithm 2 is used to transform the given Resource Constraints (RC) to *Timing Overhead*. *Timing Overhead* is the additional number of cycles by which the designer is willing to extend the program execution in order to optimize number of resources used. For each constrained resource r, the estimation algorithm (Section 4.4) computes the time overhead, d_c, (in number of cycles) if the specified upper bound of resource instances is used. The smallest of all values for d_c becomes a *Timing Overhead*. The *Timing Overhead* is a

Algorithm 2 Compute Time Overhead

```
in: $Resource\ Constraints$
in: $Histogram\ for\ each\ Constrained\ Component$
out: $Timing\ Overhead$
for all $r \in ResourceConstraints$ do
  $r.new\_num = Constraint(r)$
  $d_c = Estimate(r.new\_num)$
  if $d_c \leq TimingOverhead$ then
    $TimingOverhead = d_c$
  end if
end for
```

maximum overhead that the number of instances of an unconstrained resource may cause in order to be included in the final data path.

In our example, as per the Equation 1, the upper bound is two and if we assume that there is only one resource constraint specified, both d_c and *Timing Overhead* are equal to one cycle (Figure 4). For all other, unconstrained resources, the number of instances needs to be sufficient so that the given basic block executes in maximum seven cycles (six for the original schedule and one for *Timing Overhead*).

The estimation is done for a single resource type at a time and therefore the input is a set of histograms for that resource for selected basic blocks. The task of estimation step is to quantify the effect of change in number of instances of the resource to the application execution time and resource. In order to do so, we compute the number of extra cycles (d_c) that is required to execute a single basic block with the desired number of units (*NewNumber*) using 'Spill' algorithm (Algorithm 3).

Algorithm 3 Spill

```
in: $Histogram\ (H)\ for\ a\ Resource\ r$
in: $Number\ of\ Instances : NewNum$
out: $d_c$ //in number of cycles
for all $X = cycle \in H$ do
  $CycleBudget = NewNumber - X.InUse$;
  if $CycleBudget \geq 0 \&\& DemandCounter \geq 0$ then
    $CanFit = \mathbf{min}(CycleBudget, DemandCounter)$
    $DemandCounter += CanFit$
  else
    $DemandCounter += CycleBudget$
  end if
end for
$d_c = \lceil DemandCounter/NewNumber \rceil$
return $d_c$
```

We keep a counter (*Demand Counter*) of operations/data transfers that were originally scheduled for the execution in the current cycle on an instance of the resource r, but could not possibly be executed in that cycle with the *NewNumber* of instances. For example, in both cycles 3 and 5 (in bottom of the Figure 4) there is one operation (shown in dashed lines) that can not be executed if only two adders are used. Those operations need to be accounted for by the *Demand Counter*.

In each cycle, we compare the number of instances in use in a current cycle ($X.InUse$) to the *NewNumber*. If the number in the current cycle is greater, the number of 'extra' instances is added to the *Demand Counter*, counting the number of operations/transfers that would need to be executed later. On the other hand, if the number in the current cycle is less then the *NewNumber*, there are available resources that may execute the operations/transfers that were previously accounted for with *Demand Counter*. In the bottom of Figure 5 the available resources are shown in yellow and the 'postponed' execution of 'extra' operations is shown by arrows. The 'Spill' algorithm models in this way the delayed execution of all 'extra' operations. After going through all cycles in a given block, the *Demand Counter* equals to the number of operations that need to be executed during the additional cycles d_c.

The 'Spill' algorithm uses only statically available information and provides the overhead for a single execution of a given basic block. In order to estimate the resulting performance, we incorporate execution frequencies in the estimation. The estimated total execution time equals sum of products of block's d_c and block's frequency for each block selected for the optimization. We must note that this method does not explicitly account for interference while changing the number of instances of other resources than the specified ones.

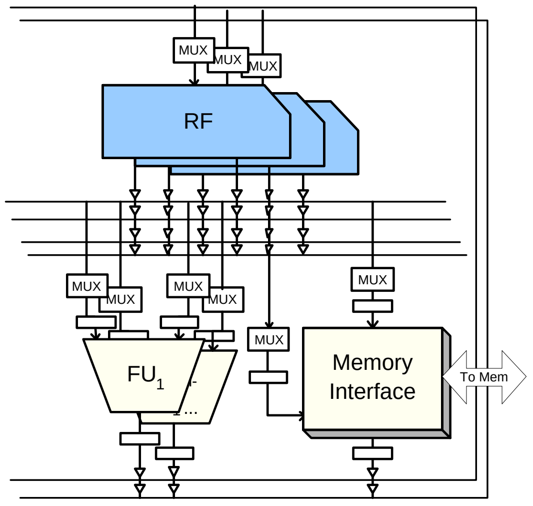


Figure 6. An Example of Extracted Data path Model.

4.5 Balancing Instances of Unrestricted Components

We assume that it is acceptable to trade off certain percentage of the execution time in order to reduce number of used resources (hence to reduce area and power and increase component utilization).

Therefore, we select a subset of all possible designs that contains all candidate designs, to be estimated. The candidate designs are created as follows:

- set the number of instances for the constrained resources to their upper bound specified by the constraints
- set the number of instances of unconstrained resources (functional units, buses, storage elements and their ports) to the same values as in the Initial Data path
- assume the same connectivity as in the Initial Data path
- randomly select a resource type and have its number of instances varied.

For the example depicted in Figure 5, where a multiplier is selected to have its number of instances varied, there will be two candidate designs created: with one and with two multipliers. The candidate design with no multipliers would not make sense, since there will be no units[1] to perform the operations that were originally performed by the multiplier. Also, there is no need to consider three multipliers, since two already satisfy the constraints.

The Algorithm 4 is used to create candidate designs and estimate their overhead. The $Update$ function assigns an average value as a starting value to resource *r.candidate_num*. The estimation returns the number of *Overhead* cycles. If the estimated *Overhead* is greater than $Timing\ Overhead$, the value of *r.candidate_num* is incremented; otherwise it is decremented. When the number of instances is determined, and the resource type gets marked as *tried*. The refinement continues by selecting the next resource type. It may happen that even if we increase the *r.candidate_num* of instances of some type, the *Timing Overhead* can not be met. The algorithm then backtracks to the minimal *r.candidate_num* that causes violation, marks the component type as *tried* and reports the *'best effort'* design.

In the simple case, shown in the Figure 5 if the Timing Overhead is 1 cycle, having 2 units would deliver required performance. Since having 1 units also results in the acceptable overhead (also 1 cycle overhead), the algorithm chooses the smaller number.

[1]This is due to the matching heuristics (Algorithm 1)

Algorithm 4 Compute Number of Instances

```
in: $AllowedTimingOverhead$
out: $SelectedNumberofInstancesofUnrestrainedComponent$
//For all Unrestrained Components
for all $r =$ rnd $(AllResources \backslash ResourceConstraints)$ do
   label
   $r.candidate\_num = Update(r)$
   $CreateNewDesign$
   $Overhead = Estimate(r.candidate\_num)$
   if $(Overhead \geq TimingOverhead)$ then
      goto label
   end if
end for
$SelectedNumber = r.candidate\_num$
return $SelectedNumber$
```

4.6 Component Allocation and Net-list Creation

In order to allocate the final data path, we use the same heuristics as described in 3. Previously, during the initial allocation, the operands from the code were matched with the components from the library to determine the type of functional unit. Here the functional unit type is inherited from the Initial Data path, and the parameters (such as the number of instances, ports, registers) are specified by the outcome of the Estimation (Section 4.4). Based on the connectivity statistics, the tool decides to provide full or limited connectivity. The full connectivity scheme is used in Initial Data path as described in Section 3. In limited connectivity scheme, we reduce number of connections from register file's output ports to the source buses, and we connect only one bus to one output port. Please note that the tool actually replaces busses with muxes in net list in order to comply with synthesis guidelines for state of the art synthesis tools. Also in order to decrease area and propagation delay, wherever applicable, the tool removes multiple levels of multiplexers that would occur by simply replacing buses with multiplexers.

Table 1. Parameter Values and Code Size (LOC).

Bench.	(P_l,P_f,P_{fl})[%]	LoC	Gen. T [sec] Non-pipe	Pipe
bdist2	(60,50,45)	61	0.2	0.8
Sort	(80,60,45)	33	0.1	0.1
dct32	(18,65,50)	1006	1.3	2.3
Mp3	(30,55,50)	13898	15.6	42.6
inv/forw 4x4	(50,45,55)	95	0.2	0.8

5 Results

We implemented the IDp Extraction and the Data path Optimization in C++. We used programmable controller unit, NISC compiler ([7, 20]) to generate schedule and Verilog generator ([9, 10]) for translating architecture description from ADL to Verilog. For synthesis and simulation of the designs, we used Xilinx ISE 8.1i and ModelSim SE 6.2g running on a 3GHz Intel Pentium 4 machine. The target implementation device was a Xilinx Virtex II FPGA xc2v2000 in FF896 package with speed grade -6. The benchmarks used were *bdist2* (from MPEG2 encoder), *Sort* (implementing bubble sort), *dct32* (from MP3 decoder), *Mp3* (decoder) and *inv/forw 4x4* (functions *inverse4x4* and *forward4x4* are amongst top five nost frequently executed functions from H.264). Profiling information was obtained manually.

Table 1 shows input parameters, benchmark length and generation time. Parameters P_l, P_f, P_{fl} are defined in Section 4.2. We decided on parameter values using the profiling information. The selected values of parameters ensure that blocks that affect the execution time the most are selected for optimization. The following column shows the number of lines of the C code (LoC). The largest C code has 13,898 lines of code, proving the ability of the proposed approach to handle large industrial scale applications. The last two columns present average generation time for non-pipelined and pipelined designs. Even for industrial size application generation time is less than one minute.

Table 2. Difference in components and parameters between respective baseline and generated design

Bench.	Pipe.	ALU1	ALU2	ALU3	RF2x1	RF4x2	RF6x3	RF8x4	RF16x8	IDp
bdist2	N	#R=64	#R=64	#R=64	#R=64	#R=64	#R=64	#R=64	#R=64	Rf 16x8, 3 Alu, 2 Mul
	Y	#R=32	#R=32	#R=32	#R=32	#R=32	#R=32	#R=32	#R=32	Rf 16x8, 3 Alu, 2 Mul
Sort	N	#R=16	#R=16	#R=16	#R=16	#R=16	#R=16	#R=16	#R=16	Rf 16x8, 2 Alu
	Y	#R=16	#R=16	#R=16	#R=16	#R=16	#R=16	#R=16	#R=16	Rf 16x8, 2 Alu
dct32	N	Rf 4x2	Rf 6x3	Rf 8x4	-	2 Alu	3 Alu	3 Alu	3 Alu	Rf 16x8, 3 Alu, 2 Mul
	Y	Rf 4x2	Rf 4x2	Rf 6x3	-	2 Alu	2 Alu	2 Alu	3 Alu	Rf 16x8, 3 Alu, 2 Mul
Mp3	N	-	Rf 6x3	Rf 8x4	-	2 Alu	3 Alu	3 Alu	3 Alu	Rf 16x8, 3 Alu, 2 Mul
	Y	Rf 4x2	Rf 6x3	Rf 6x3	-	2 Alu	2 Alu	2 Alu	2 Alu	Rf 16x8, 3 Alu, 2 Mul
inv/forw 4x4	N	#R=16	#R=16	#R=16	#R=16	#R=16	#R=16	#R=16	#R=16	Rf 16x8, 4 Alu
	Y	#R=16	#R=16	#R=16	#R=16	#R=16	#R=16	#R=16	#R=16	Rf 16x8, 4 Alu

In this paper, we present three sets of experiments. The first set of experiments illustrates design space exploration using the automatic extraction of data path from application C code. The second set of experiments compares our selection algorithm to manual selection of componenets from C code. The last set of experiments compares the presented extraction technique to HLS and manual design in order to establish quality of generated designs.

5.1 Interactive Design Exploration

Results of the exploration experiments are shown in Figures 7, 8 and 9. Used baseline data path architectures are MIPS-style manual designs (pipelined and non-pipelined) [10] with an ALU, a multiplier, two source and one destination bus and a 128-entry register file with one input and two output ports. Only for the *Mp3* application we have added a divider unit to this processor for comparison with the generated data path. In order to perform fair comparison the size of storage elements has been customized for every application such that the resources (area) are minimized. Also, for comparison with automatically generated pipelined design, the *pipelined version* of manual design was used as a baseline. In-house compiler is used to create schedule for all baseline and generated data paths. This guarantees that the execution time depends on the data path architecture and does not depend on the compiler optimizations.

While exploring different designs for selected applications we specified the resource constraints on the number of ALUs and number of output and input ports of register file (RF). The tool extracts a data path from the C code such that it complies to the specified constraint and resulting data paths are named as:

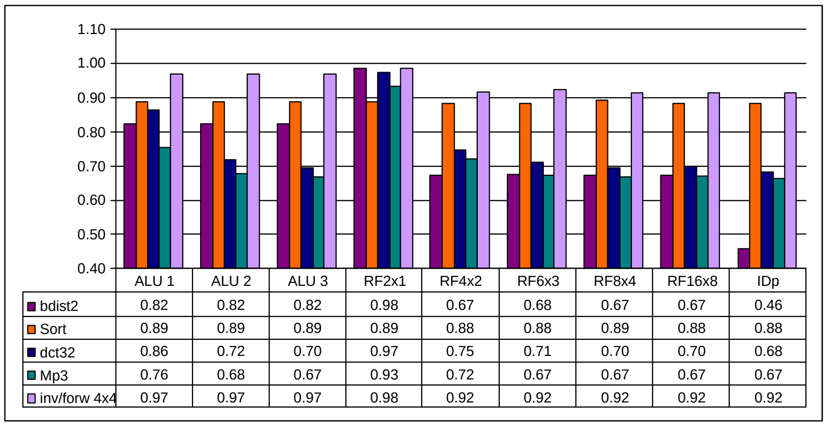
- *ALU N*, where $N \in \{1,2,3\}$ is specified number of ALUs
- *RFOxI*, where $(O,I) \in \{(2,1), (4,2), (6,3), (8,4)\}$ are the number of output and input ports.

In case of data path denoted by *RFOxI*, *two* resource constraints were used to generate the design: one for the number of output and the other for the number of input ports while all the remaining elements (like functional units, memories and connectivity) are decided by the tool as described in Section 4.

Fig. 7 shows the number of execution cycles for generated architectures normalized to the number of cycles for the baseline architecture. This graph includes two additional data paths that are generated only to illustrate tool behavior and to explore theoretical limits of the used data path model (Fig. 4.6). Those additional configurations are:

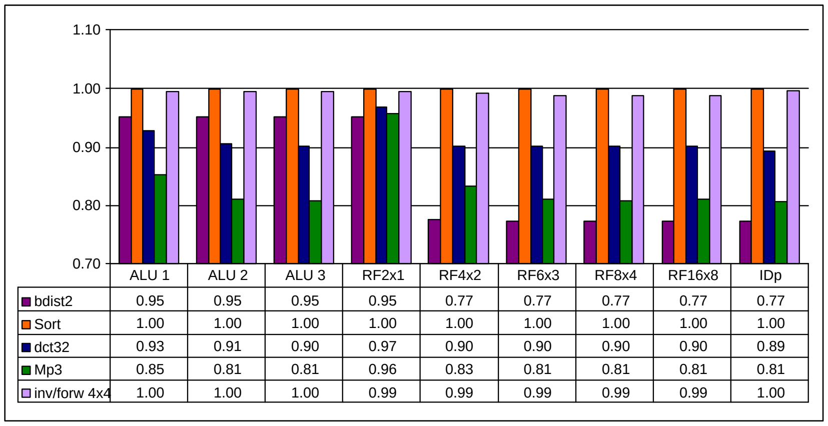
- *RF16x8*: a configuration that was generated using RF 16x8 constraint
- *IDp*: an Initial Data path

Table 2 summarizes generated architectures for all the configurations that are presented in this paper. Each benchmark and each configuration have a non-pipelined and a pipelined architecture generated, and those are presented in rows marked with



	ALU 1	ALU 2	ALU 3	RF2x1	RF4x2	RF6x3	RF8x4	RF16x8	IDp
bdist2	0.82	0.82	0.82	0.98	0.67	0.68	0.67	0.67	0.46
Sort	0.89	0.89	0.89	0.89	0.88	0.88	0.89	0.88	0.88
dct32	0.86	0.72	0.70	0.97	0.75	0.71	0.70	0.70	0.68
Mp3	0.76	0.68	0.67	0.93	0.72	0.67	0.67	0.67	0.67
inv/forw 4x4	0.97	0.97	0.97	0.98	0.92	0.92	0.92	0.92	0.92

(a) Non-pipelined



	ALU 1	ALU 2	ALU 3	RF2x1	RF4x2	RF6x3	RF8x4	RF16x8	IDp
bdist2	0.95	0.95	0.95	0.95	0.77	0.77	0.77	0.77	0.77
Sort	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
dct32	0.93	0.91	0.90	0.97	0.90	0.90	0.90	0.90	0.89
Mp3	0.85	0.81	0.81	0.96	0.83	0.81	0.81	0.81	0.81
inv/forw 4x4	1.00	1.00	1.00	0.99	0.99	0.99	0.99	0.99	1.00

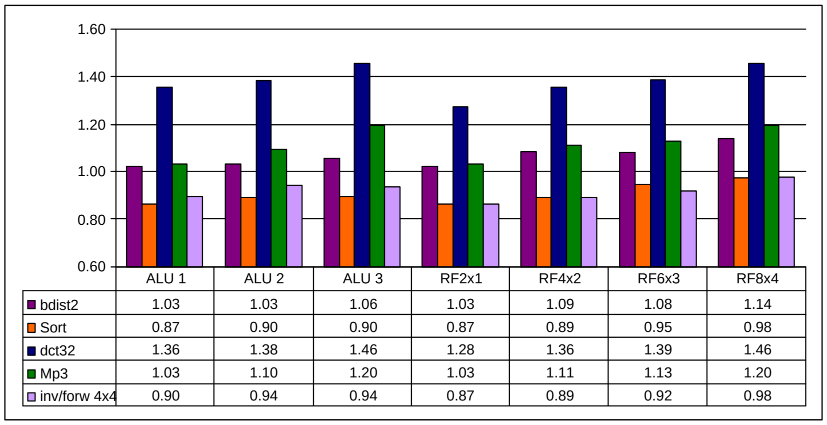
(b) Pipelined

Figure 7. Relative number of execution cycles on data paths generated for different architecture

N and Y in the column ‘Pipe.’ The table lists the difference from the corresponding baseline architecture. For example: ‘#R=64’ means that the generated data path has 64 registers in register file, ‘Rf 4x2’ means that there is a register file with 4 output and 2 input ports and ‘-’ means that there is no change to the baseline data path parameters.

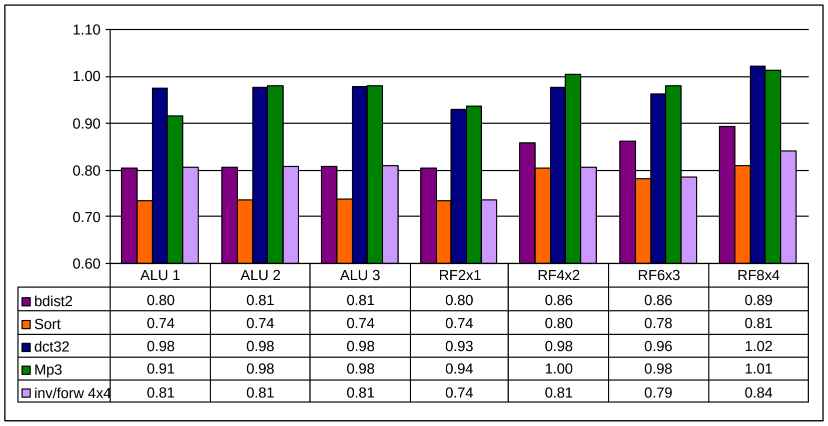
For generated non-pipelined data paths (Fig. 7(a)) normalized execution cycles range from 0.98 to 0.46. All of the benchmarks experience only a small improvement for *RF2x1* configuration because this configuration is the most similar to the baseline. Also, the number of cycles is not exactly the same but slightly reduced. This effect is due to replacing of buses in architecture specification with multiplexers which allows for more efficient handling by the compiler. This effect is particularly emphasized in case of *bdist2*. For this benchmark the improvements are small (0.82) for all ALU configurations and may be attributed to the effect of explicit multiplexer specification that results in more efficient compiler handling and shorter prologue/epilogue code. We see the further reduction to 0.67 for *RF4x2* configuration which exploits the parallelism of operations executed on different units. No further improvement is seen for further increase in the number of register file ports. However, execution on the *IDp*, which has two ALU and two multiplier units, experiences additional improvement in number of cycles to 0.47.

Benchmark *Sort* is sequential in nature and therefore it does not experience significant improvement regardless of the number of ALUs or register file ports that are introduced. Both benchmarks *dct 32* and *Mp3* have abundance of available parallelism. The *dct32* benefits the most from having two ALUs (*ALU2* - 0.72) and increase in number of ports or adding more



	ALU 1	ALU 2	ALU 3	RF2x1	RF4x2	RF6x3	RF8x4
🟪 bdist2	1.03	1.03	1.06	1.03	1.09	1.08	1.14
🟧 Sort	0.87	0.90	0.90	0.87	0.89	0.95	0.98
🟦 dct32	1.36	1.38	1.46	1.28	1.36	1.39	1.46
🟩 Mp3	1.03	1.10	1.20	1.03	1.11	1.13	1.20
🟪 inv/forw 4x4	0.90	0.94	0.94	0.87	0.89	0.92	0.98

(a) Non-pipelined



	ALU 1	ALU 2	ALU 3	RF2x1	RF4x2	RF6x3	RF8x4
🟪 bdist2	0.80	0.81	0.81	0.80	0.86	0.86	0.89
🟧 Sort	0.74	0.74	0.74	0.74	0.80	0.78	0.81
🟦 dct32	0.98	0.98	0.98	0.93	0.98	0.96	1.02
🟩 Mp3	0.91	0.98	0.98	0.94	1.00	0.98	1.01
🟪 inv/forw 4x4	0.81	0.81	0.81	0.74	0.81	0.79	0.84

(b) Pipelined

Figure 8. Relative cycle time on data paths generated for different architecture

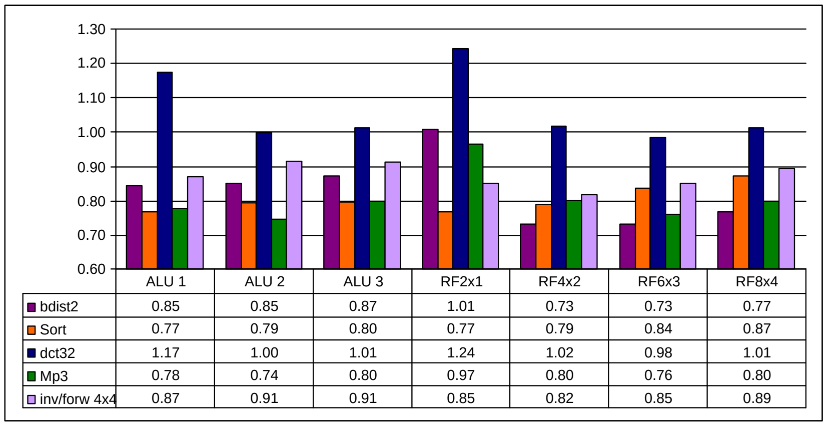
units (in *IDp*) contributes by only 4% of additional improvement. Similarly, *Mp3* executes in 0.68 of the number of baseline execution cycles for *ALU2* configuration. Note that specifying two ALUs as resource constraint for both benchmarks results in an increase in the number of RF ports and buses: since both instances of ALU and one instance of multiplier are significantly utilized, the resulting configurations have RF 6x3 and full connectivity scheme. On the other hand, both benchmarks suffer from significant increase of prolong/epilogue code which sets back the savings in number of cycles that are obtained by the 'body' of the benchmark. Adding more ALUs does not help in case of benchmark *inv/forw 4x4*. The benchmark benefits the most from additional register file ports, because this configuration exposes limited parallelism beetween the operations that execute on functional units of different type.

As for the pipelined configurations, shown in Fig. 7(b), across all the benchmarks maximum reduction in the number of execution cycles for generated data paths (0.77) is less than a maximum reduction for the non-pipelined designs since the pipelining itself exploits some degree of available parallelism. In case of *bdist2* there is no improvement with increased number of ALUs since the tool allocates single *RF2x1*. Same as for the non-pipelined configurations, the minimal normalized number of cycles is reached for *RF4x2* due to the increased simultaneous use of ALU and multiplier. On the other hand, benchmark *Sort* does not change the number of execution cycles since pipelining takes advantage of already limited available parallelism.

For *dct32* the tool allocates RF 4x2, RF 4x2 and RF 6x3 (together with sufficient connectivity resources) for configurations

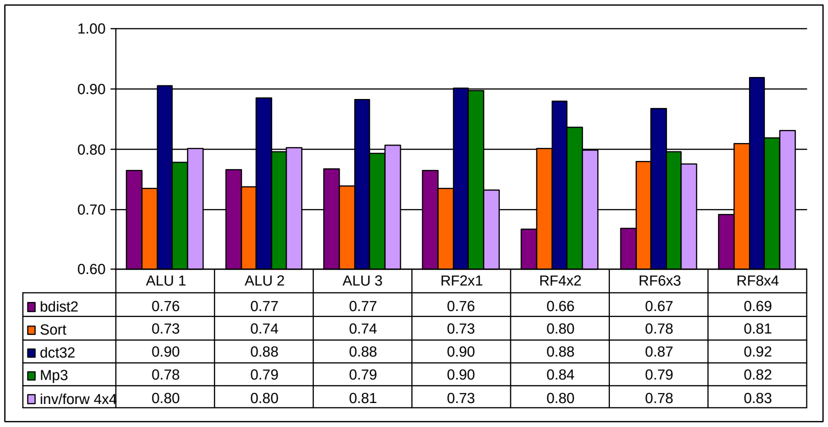
ALU 1, *ALU 2* and *ALU 3* respectively, where for *Mp3* it allocates RF 4x2, RF 6x3 and RF 6x3. The most of the execution cycle reduction is brought by increase in number of register file ports in case of configuration *ALU 1*. Increasing both number of ALUs and ports brings down the normalized cycles only by 0.02 and 0.04 down to 0.90 and 0.81 for *dct32* and *Mp3*, respectively. For all the RF configurations, both benchmarks have the same trend for allocation: the tool recognizes a potential for adding more ALUs and therefore two ALUs are allocated for all of them, except for *RF16x8* configuration of *dct32* where three ALUs are allocated. One would expect the tool would allocate more units for *RF8x4* and *RF16x8*. However, the data dependencies limit concurrent usage of more units than allocated. The results for *IDp* illustrate this: even though *IDp* has three ALUs and two multipliers, further reduction in number of normalized cycles is only by 0.01 to 0.02 for *dct32* and *Mp3*, respectively. Similarly to *Sort*, *inv/forw 4x4* has almost no improvement if the number of instances increases.

Fig. 8 show normalized cycle time for non-pipelined and pipelined automatically generated designs. We observed the cycle time in order to explain the total execution time of each benchmark. Current version of the tool does not take into account post synthesis results. However, we believe that this feature is crucial for DFM and are currently working on incorporating pre-layout information in data path optimization.



	ALU 1	ALU 2	ALU 3	RF2x1	RF4x2	RF6x3	RF8x4
🟪 bdist2	0.85	0.85	0.87	1.01	0.73	0.73	0.77
🟧 Sort	0.77	0.79	0.80	0.77	0.79	0.84	0.87
🟦 dct32	1.17	1.00	1.01	1.24	1.02	0.98	1.01
🟩 Mp3	0.78	0.74	0.80	0.97	0.80	0.76	0.80
🟪 inv/forw 4x4	0.87	0.91	0.91	0.85	0.82	0.85	0.89

(a) Non-pipelined

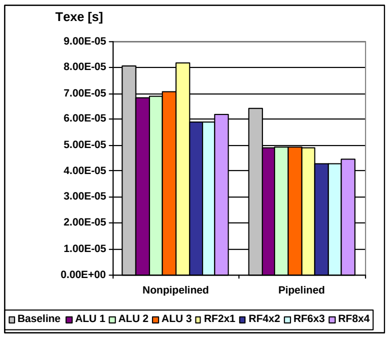


	ALU 1	ALU 2	ALU 3	RF2x1	RF4x2	RF6x3	RF8x4
🟪 bdist2	0.76	0.77	0.77	0.76	0.66	0.67	0.69
🟧 Sort	0.73	0.74	0.74	0.73	0.80	0.78	0.81
🟦 dct32	0.90	0.88	0.88	0.90	0.88	0.87	0.92
🟩 Mp3	0.78	0.79	0.79	0.90	0.84	0.79	0.82
🟪 inv/forw 4x4	0.80	0.80	0.81	0.73	0.80	0.78	0.83

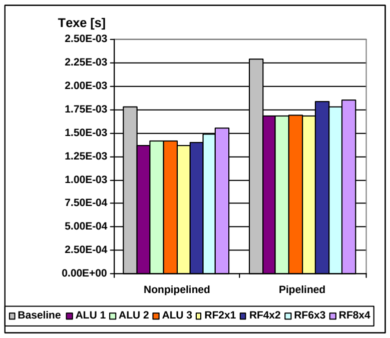
(b) Pipelined

Figure 9. Relative execution time on data paths generated for different architecture

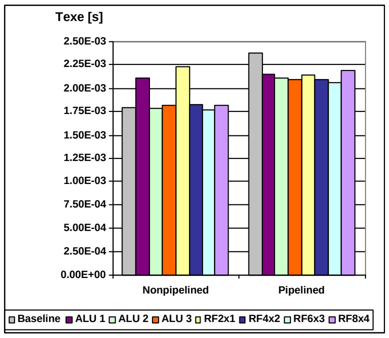
Results for normalized cycle time for designs are intuitive: as complexity of generated data path increases, so does the cycle time. For non-pipelined designs (Fig. 8(a)), designs for all benchmarks except *Sort* have larger cycle time than for corresponding baseline. The main contributor to cycle time length is register file: as the number of ports increase, the decoding logic increases and so does the cycle time. In case of *Sort* cycle time is lower because of reduction of the register file size. For non-pipelined configurations the normalized cycle time ranges from 0.85 to 1.46. Pipelined configurations (Fig 8(b)) uniformly have smaller or equal cycle time as the baseline configuration. For each benchmark, there is almost no difference in cycle time across all *ALU* configurations. *Mp3* is the only benchmark that has significantly lower normalized cycle time for *ALU1* configuration (0.91) than for remaining two *ALU* configurations (0.98). *RF* configurations experience the increase in cycle time with increase in complexity. For *RF6x3* in case of *Sort*, *dct32*, *Mp3* and *inv/forw 4x4* there is a small decrease in cycle time comparing to *RF4x2* configurations because the synthesis too manages to decrease combinational delay of interconnect.



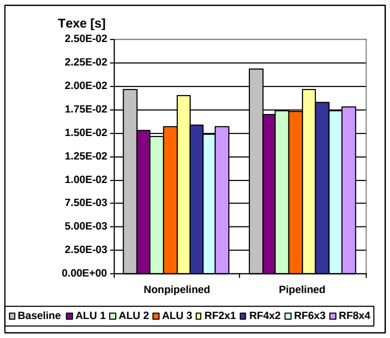
(a) bdist2



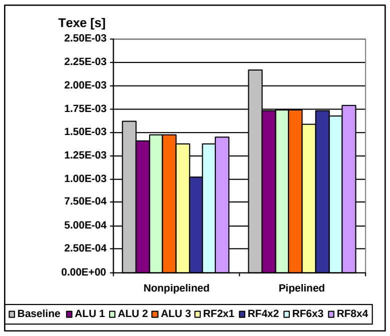
(b) Sort



(c) dct32



(d) Mp3



(e) inv/forw 4x4

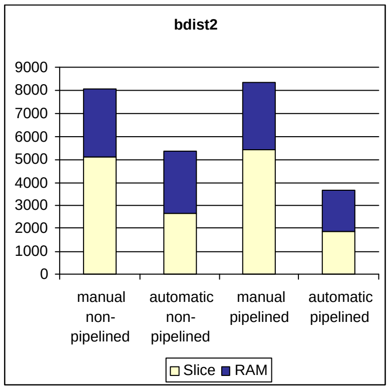
Figure 10. Total execution time on generated data paths

Fig. 9 shows normalized total execution time. Across all configurations and all benchmarks, except all non-pipelined configurations for *dct32*, total execution time has been reduced. Non-pipelined *dct32* experiences increase in execution time for all but *ALU2* configuration: the reduction in number of cycles is not sufficient to offset the large increase in the cycle time. The reduction in number of cycles is less than expected because of explosion of prologue/epilogue code. The non-pipelined configurations reduce the execution time up to 0.73, 0.77, 1.00, 0.74 and 0.82 for *bdist2*, *Sort*, *dct32*, *Mp3* and *inv/forw 4x4*, respectively. Normalized execution times for all non-pipelined configurations, except for the *Sort*, are greater than the corresponding normalized number of cycles. The *Sort* has further decrease in execution time due to significant cycle time reduction (resulting from 'minimized' data path comparing to the baseline). Furthermore, for *dct32* and *Mp3*, that perform the best for *ALU2*, several other configurations have minimum normalized number of cycles. Pipelined configurations uniformly experience smaller normalized execution time comparing to the non-pipelined. The minimums are 0.66, 0.73, 0.88, 0.79 and 0.73 for *bdist2*, *Sort*, *dct32*, *Mp3* and *inv/forw 4x4*, respectively. For all applications, each normalized execution time is smaller than the corresponding normalized number of execution cycles. Furthermore, the configurations that perform in minimal time are the same as the one that perform in minimal number of cycles.

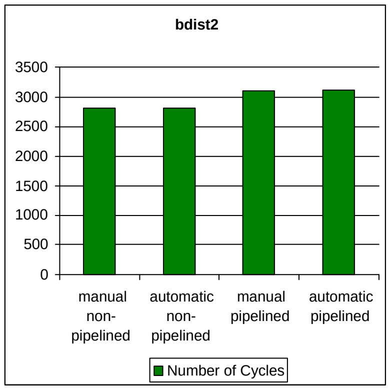
In order to find a data path with with a minimum execution time and the best configuration we plot for each benchmark (absolute) execution time in Fig. 10. The leftmost bar shows the execution time on a baseline architecture. The best implementation for *bdist2* is pipelined *RF4x2*. *RF6x3* has only slightly longer execution time, but since it is uses more resources, it is less desirable (recommendable) choice. Benchmark *Sort* benefits from reduction of resources and therefore the best configuration is *ALU1*. For this benchmark, all of the pipelined configurations perform worse than corresponding non-pipelined. Benchmark *dct32*, despite having plethora of available parallelism performs good only for non-pipelined *Baseline*, *ALU2* and *RF6x3* configurations. The pipelined configurations do not perform as well as non-pipelined. To improve on the current generated pipelined architectures, we may consider use of multi-cycle and pipelined functional units which may reduce the cycle time. Furthermore, if there is only single function to be performed on the generated hardware module, both prologue and epilogue code may be eliminated and the speedup of 'parallel' architectures would increase. Here we presented the results for all the applications with prologue/epilogue code because we believe that the application execution needs to have data received and sent proceeding and following the execution of the benchmark body. Therefore, benchmark is a function that needs to be called and therefore the prologue and epilogue code are required. In this case, the number of registers that need to be stored and restored, and hence the length of prologue and epilogue code, needs to be estimated. Non-pipelined designs for *Mp3* perform better than pipelined, for the same reason. Overall, the best design would be for *ALU2* configuration with 32% performance imptovement over the baseline.

Table 3. Comparison between components and parameters of manual and automatically generated design

		Automatic	
Bench.	Manual	Non-pipe	Pipe
bdist2	#R=32, Rf 8x4, 4 Alu, 1 Mul	#R=64, Rf 4x2, 1 Alu, 1 Mul, 1 Comp	#R=32, Rf 4x2, 1 Alu, 1 Mul, 1 Comp
Sort	#R=32, Rf 4x2, 1 Alu	#R=32, Rf 2x1, 1 Alu, 1 Mul, 1 Comp	#R=32, Rf 2x1, 1 Alu, 1 Mul, 1 Comp
dct32	#R=48, Rf 8x4, 4 Alu, 2 Mul, 1 Comp, 2 Adders	#R=128, Rf 8x4, 3 Alu, 1 Mul, 1 Comp	#R=128, Rf 4x2, 2 Alu, 1 Mul, 1 Comp
Mp3	#R>16, Rf 16x8, 4 Alu, 8 Mul 1 Or, 1 Comp, 1 NotEq Comp, 1 Div	#R=128, Rf 8x4, 3 Alu, 1 Mul, 1 Comp, 1 Div	#R=128, Rf 2x1, 1 Alu, 1 Mul, 1 Comp, 1 Div
inv/forw 4x4	#R=32, Rf 8x4, 4 Alu, 1 Comp	#R=16, Rf 4x2, 1 Alu, 1, Mul, 1 Comp	#R=16, Rf 2x1, 1 Alu, 1 Mul, 1 Comp

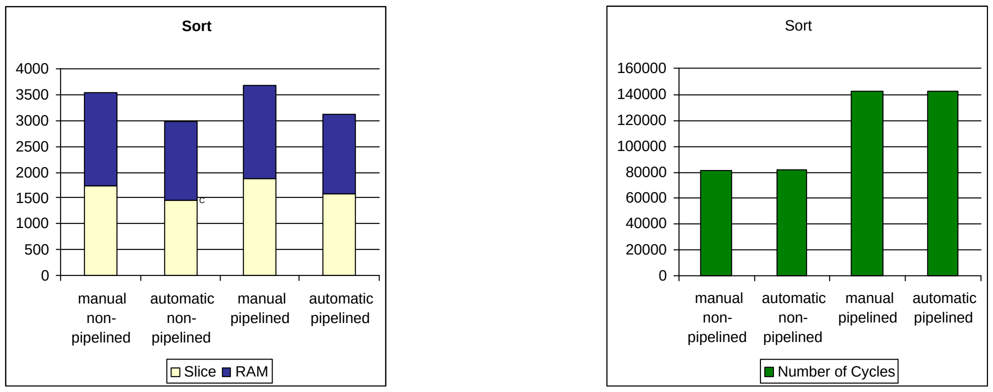


(a) Cost



(b) Performance

Figure 11. bdist2: Number of slices, number of RAMs and number of cycles for manually selected and automatically generated data paths



(a) Cost (b) Performance

Figure 12. Sort: Number of slices, number of RAMs and number of cycles for manually selected and automatically generated data paths

5.2 Selection Algorithm Quality

Table 3 shows the comparison of manually designed architectures and the automatically generated ones. The manual designs were created by computer engineering graduate students. The students were asked to select the components for the templatized data path, as the one in Figure 6, based on the application C code. Running and profiling the code on the host machine with the same input as used for the automatic generation data was allowed.

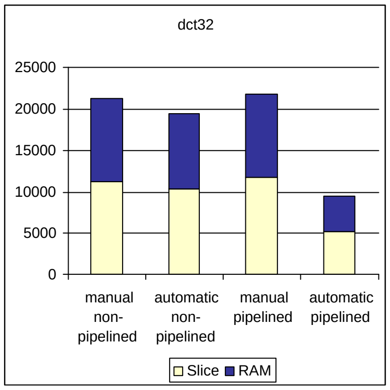
There is only one column for manual designs in the Table 3 because the designers had *the same* component/parameter selection for non-pipelined and for pipelined data paths. However, our experiments in Section 5.1 show that often, there is less resources required for pipelined configurations. Such examples are configurations *ALU2*, *ALU3*, *RF 6x3* and *RF 8x4* for *dct32* in Table 2. The non-pipelined and pipelined configurations presented in Table 3 are the one that have the smallest number of execution cycles for the given benchmark, as seen in the Figure 7.

It is interesting to notice that for manual designs, in most cases, the number of instances and parameters of selected register files and functional units outnumbers the one in the best generated architectures. For example, for benchmark *bdist2* manual designer anticipated use of four ALUs. The non-pipelined IDp for benchmark *bdist2* needs only three and pipelined IDp only one ALU, which shows that the designer overestimates the number of ALUs. Also, the optimal automatically generated data path uses only one ALU in both non-pipelined and pipelined case. However, for this benchmark, the designer underestimated register file size: in case where there are more functional units, more operations may be perform in parallel and therefore there will be more operands/registers required. The tendency to allocate manually more resources than actually required may be explain the best on the example of function *inverse4x4* from *H.264*, shown in Algorithm 5.

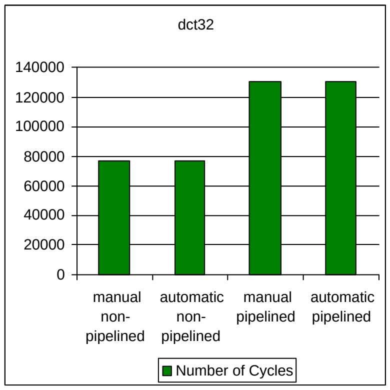
Algorithm 5 Part of *inverse4x4* C code

```
...
t0 = *(pblock++);
t1 = *(pblock++);
t2 = *(pblock++);
t3 = *(pblock );

p0 = t0 + t2;
p1 = t0 - t2;
p2 = SHIFT(t1, 1) - t3;
p3 = t1 + SHIFT(t3, 1);
...
```



(a) Cost



(b) Performance

Figure 13. dct32: Number of slices, number of RAMs and number of cycles for manually selected and automatically generated data paths

The designer allocates four ALUs based on an observation that code in lines 7, 8, 9 and 10 is not dependent and an assumption that once all if the operations from line 2 to line 5 are completed, the entire block of lines 7 to 10 will be executed at the same time. However, code in lines 2 to 5 has data dependencies, requires sequential execution and performs memory access. Therefore, it makes sense to compute expressions in line 7 and line 8 as soon as t0 and t2 are available. Hence, no need for 4 ALU in the data path.

Similarly, for *dct32*, data dependencies are not ‘visible’ from C code. Therefore, the designer allocates four ALUs, two multipliers, a comparator and two adders. IDp for *dct32* has only three ALUs, two multipliers and a comparator even though it has a register file RF 16x8. IDp configuration performs in 0.68 of the baseline, which is only 2% better than configurations *ALU3*, *RF8x4* and *RF16x8* that have less resources. The number of registers in the register file is computed based on the number of units (eight without the comparator), the fact that each unit has two inputs and one output, and assumption that for each source/destination the data memory will be used twice. Therefore:

$$\#R = 8 \times (2+1) \times 2 = 48 \tag{5}$$

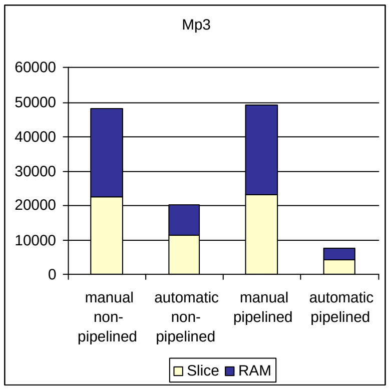
i.e. the designer decides on 48 registers. Practically, with these many units, there are more than 48 registers required for temporary variables, if we want to avoid access to memory for fetching data and storing results.

Manual selection of components and parameters for *Mp3* shows the same properties: the number of functional units was overestimated, the number of registers in the register file was underestimated and the pipelining was selected after the decision on units had been made. The designer profiled the application and found that among all computationally intensive functions, function *synth_full* contributes 35% to total execution. The designer identified eight multiplications and four additions that may be executed in parallel in this function. Also, only the lower bound for the number of registers in the register file was specified.

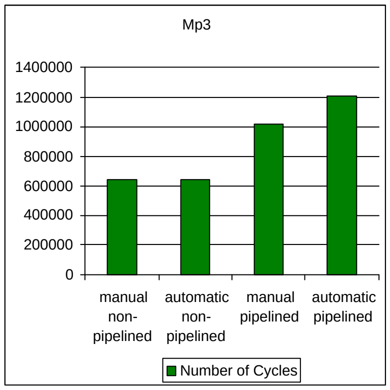
In order to better understand cost/performance trade-off for manually and automatically generated data paths we defined a total cost of a design C_{design} as a sum of slices and a sum of RAMs for all the selected components.

$$C_{design} = \sum_{components} slice + \sum_{components} RAM \tag{6}$$

We synthesized all available components and assigned cost in terms of slices and RAMs. We generated both non-pipelined and pipelined versions of manual design, so that we can perform fair comparison of cost and performance. We assumed that when pipelining was selected all inputs and outputs of functional units have a pipeline register (uniform pipelining, such as shown in Figure 6). Cost of pipeline registers was added to the total cost of pipelined data path designs. The performance is measured in the number of execution cycles, since neither the designers nor the tool were given any synthesis information as an input.



(a) Cost



(b) Performance

Figure 14. Mp3: Number of slices, number of RAMs and number of cycles for manually selected and automatically generated data paths

Figures 11, 12, 13, 14 and 15 show set of cost/performance graphs for all presented benchmarks. All automatically generated designs have significantly lower cost comparing to the ones manually derived from C code. Relative to the corresponding manual designs, the automatically generated ones have from 7.41% (*dct32* non-pipelined) to 82.13% (pipelined *inv/forw 4x4*) less slices and from 8.22% (non-pipelined *bdist2*) to 87.46% (pipelined *Mp3*) less RAMs. As seen in the performance graphs, overhead in the number of cycles is negligible for all designs except for the above mentioned pipelined *Mp3*. *Mp3* has 18.84% overhead, but 81.50% and 87.46% less slices and RAMs, which according to us is a reasonable trade-off. For all the remaining designs, overhead of the number of cycles ranges from 0.00% to 0.68% relative to the corresponding manual design.

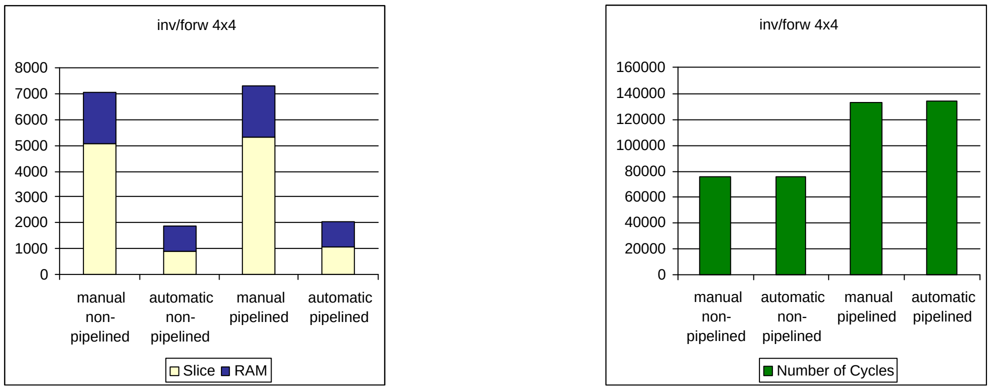
This experiment showed that translating C code into simple hardware design is a non-trivial process. Selecting components and their parameters, tracking data and control dependencies, and estimating operation sequencing and available parallelism based on high level description results in underutilization of data path elements. On the other hand using our tool, within the same time, a designer may explore many different alternatives and create working solution that satisfies his or her needs.

5.3 Design Refinement Quality

Fig. 16 plots values for the number of cycles (No.cycle), clock cycle time (Tclk), total execution time (Texe), number of slices and bRAMs (Slice and BRAM) for several different data paths for *dct32* benchmark. All the values have been normalized to the corresponding values of a *manually designed* data path for the same application. Note that the same C code has been used as a starting point for all designs, including manual. The graphs show following data paths:

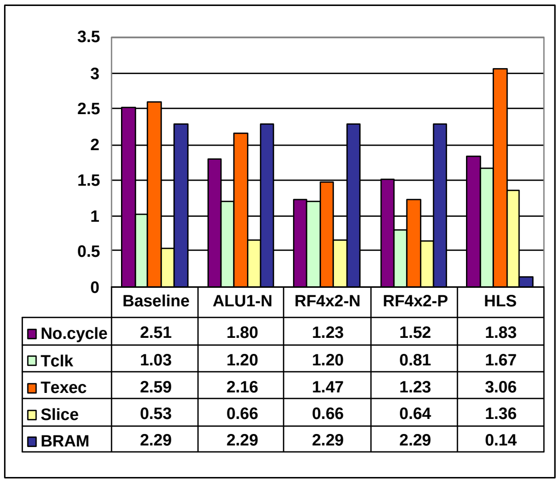
- *Baseline* - corresponds to a pipelined version of a baseline design for the *dct32* used in 5.1
- *ALU1-N* and *RF4x2-N* - generated *non-pipelined* data paths for constraints ALU 1 and RF 4x2, respectively
- *RF4x2-P* - generated *pipelined* data path for constraint RF 4x2
- *HLS* - a design generated by academic high level synthesis tool [21]

To alleviate different assumptions of different tools and designers for wrapping the function by send/receive primitives, we present here the results for the body of the *dct32* function, contrary to experiments in 5.1. The manual implementation has been designed by third-party RTL designer [19]. It is important to notice, that the largest normalized value accross all metrics is 3.3 times the corresponding metric of the manual design. Hence, none of the compared metrics are several orders of magnitude larger than the manual design. The overhead of number of cycles for generated designs range from 23% (i.e 1.23 on the graph) to 80% of the manual design, while cycle time experiences from 25% (0.85 in the figure) speedup to 25% (1.25 in the figure) slowdown. The best generated design *RF4x2-P* has 1.23 times longer execution time comparing to the



(a) Cost (b) Performance

Figure 15. inv/forw 4x4: Number of slices, number of RAMs and number of cycles for manually selected and automatically generated data paths



	Baseline	ALU1-N	RF4x2-N	RF4x2-P	HLS
🟪 No.cycle	2.51	1.80	1.23	1.52	1.83
🟩 Tclk	1.03	1.20	1.20	0.81	1.67
🟧 Texec	2.59	2.16	1.47	1.23	3.06
🟨 Slice	0.53	0.66	0.66	0.64	1.36
🟦 BRAM	2.29	2.29	2.29	2.29	0.14

Figure 16. Performance and area comparison for *dct32*

manual. Baseline and all generated architectures have from 0.53 to 0.64 times slices and 2.29 times block RAMs (bRAMs) compared to the manual design. This is because the tool attempts to map all storage elements to bRAM on FPGA. On the other hand, the design generated by HLS tool use 1.36 times slices and only 0.14 times bRAMs due to heavy use of registers and multiplexers. The generated designs outperform the design produced by HLS tool with respect to all the metrics except the number of used bRAMs. Moreover, the average generation time for *dct32* is 2.3 seconds while it took 3 man-weeks for the manual design. The fastest extracted design has only 23% of execution overhead and a negligible generation time compared to the manual design. Hence, we believe that the proposed data path extraction from C code is valuable technique for creation an application specific data path design.

6 Conclusions and Future Work

In this paper we presented a novel problem of finding a matching data path architecture for a given application C code. We proposed a solution in a form of technique for data path architecture extraction from the application code itself. The technique is based on matching the code properties to hardware components and templates. The final data path architecture was optimized to confirm to designer's resource constraints. The proposed technique allows handling of any size of C code, controllability of the design process, fine tunning of both data path and controller separately and having an arbitrary number of components and connections. As a proof of concept, we implemented the automatic data path extraction technique and presented a series of experiments on wide range of application size, from small to industry size. Each data path architecture was generated in less than a minute allowing the designer to explore several different configurations in much less time than required for finalizing a single design. The selection algorithm and our iterative design technique lead to significant cost reduction with negligible performance degradation comparing to the corresponding designs that have been created by designers who started from C source code. We define a total cost of a design in terms of total number of slices and RAMs for all selected components, and performance in terms of number of the execution cycles. Our experiments showed that up to 82.13% of slices and 87.46% of RAMs was saved. The number of execution cycles was 18.84% more in case of a single benchmark and for the remaining benchmarks, the maximum increase in the number of cycles was 0.68%. We measured design refinement quality on an example of *dct32* for which we synthesized all the designs on an FPGA board. We also showed that the best generated data path architecture is only 23% slower, had 2.29 times more BRAMs and 0.64 times slices utilized comparing to the manual design.

The presented approach is geared toward performance optimization. Our current work includes incorporating the pre-layout information in order to optimize for area, cycle time and power. The future work includes automatic determining the best number of pipelined stages for both components and the data path as well as extracting memory hierarchy structure form a given C code. We are also interested in using data dependency informations to guide automatic functional unit chaining, data forwarding and utilization of special functional units.

7 Acknowledgments

The authors wish to thank Pramod Chandraiah for providing the Mp3 source code. Many thanks to Pramod Chandraiah, Weiwei Chen, Gunar Schirner, Lin Yang and Bin Zhang for manual designs and Roger Ang, Hansu Cho and Dongwan Shin for manual and HLS designs for *dct32*. We would also like to thank Mehrdad Reshadi and Bita Gorjiara for compiler support and Verilog generator.

References

[1] 2005. Automated Confi gurable Processor Design Flow, White Paper, Tensilica, Inc. http://www. tensilica.com/pdf/Tools_white_paper_ fi nal-1.pdf January 2005.

[2] B. Landwehr, P. Marwedel, and R. Dömer. OSCAR: Optimum Simultaneous Scheduling, Allocation and Resource Binding Based on Integer Programming. In *Proc. European Design Automation Conference*, pages 90–95, Grenoble, France, 1994. IEEE Computer Society Press.

[3] F. Brewer and D. Gajski. Chippe: A system for constraint driven behavioral synthesis. *IEEE Trans. on Computer-Aided Design*, jul 1990.

[4] S. Devadas and R. Newton. Algorithms for hardware allocation in data path synthesis. *IEEE Trans. on Computer-Aided Design*, jul 1989.

[5] 2006. Diamond Standard Processor Core Family Architecture, White Paper, Tensilica, Inc. http://www. tensilica.com/pdf/Diamond WP.pdf, October 2006.

[6] 2008. Forte Design System Cynthesizer http://www.forteds.com/products/cynthesizer.asp.

[7] D. Gajski. Nisc: The ultimate reconfi gurable component. Technical report, Technical Report TR 03-28, University of California-Irvine, October 2003.

[8] D. Goodwin and D. Petkov. Automatic generation of application specifi c processors. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2003.

[9] B. Gorjiara, M. Reshadi, P. Chandraiah, and D. Gajski. Generic netlist representation for system and pe level design exploration. In *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis*, pages 282–287, New York, NY, USA, 2006. ACM.

[10] B. Gorjiara, M. Reshadi, and D. Gajski. Generic architecture description for retargetable compilation and synthesis of application-specifi c pipelined ips. In *Proceedings of International Conference on Computer Design (ICCD) (CODES+ISSS)*, 2006.

[11] P. Gutberlet, J. Müller, H. Krämer, and W. Rosenstiel. Automatic module allocation in high level synthesis. In *Proceedings of the Conference on European Design Automation (EURO-DAC '92)*, pages 328–333, 1992.

[12] P. Marwedel. The MIMOLA system: Detailed description of the system software. In *Proceedings of Design Automation Conference*. ACM/IEEE, June 1993.

[13] 2008. Mentor Graphics Catapult Synthesis http://www.mentor.com/products/esl/ high_level_synthesis/catapult_synthesis/index.cfm.

[14] 2008. Mentor Graphics Technical Publications: Alcatel Conquers the Next Frontier of Design Space Exploration using Catapult C Synthesis http://www.mentor.com/techpapers/fulfi llment/upload/mentorpaper_22739.pdf.

[15] 2008. Mentor Graphics Technical Publications: Designing High Performance DSP Hardware using Catapult C Synthesis and the Altera Accelerated Libraries http://www.mentor.com/techpapers/fulfi llment/upload/mentorpaper_36558.pdf.
[16] 2008. Mentor Graphics Technical Publications http://www.mentor.com/training_and_services/tech_pubs.cfm.
[17] 2008. NEC CyberWorkBench http://www.necst.co.jp/product/cwb/english/index.html.
[18] P. Paulin and J. Knight. Force-directed scheduling for the behavioral synthesis of ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, jun 1989.
[19] R. Ang http://www.cecs.uci.edu/presentation _slides/ESE-BackEnd2.0-notes.pdf.
[20] M. Reshadi and D. Gajski. A cycle-accurate compilation algorithm for custom pipelined datapaths. In *International Symposium on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2005.
[21] D. Shin, A. Gerstlauer, R. Dömer, and D. D. Gajski. An interactive design environment for C-based High-Level Synthesis. In A. Rettberg, M. C. Zanella, R. Dömer, A. Gerstlauer, and F.-J. Rammig, editors, *IESS*, volume 231 of *IFIP*, pages 135–144. Springer, 2007.
[22] 2005. Stretch. Inc.: S5000 Software-Confi gurable Processors http://www.stretchinc.com/products/ devices.php.
[23] 2005. Tensilica: Xtensa LX http://www.tensilica.com/products/xtensa _LX.htm.
[24] J. Trajkovic, M. Reshadi, B. Gorjiara, and D. Gajski. A graph based algorithm for data path optimization in custom processors. In *Proceedings of 9th EUROMICRO Conference on Digital System Design*, pages 496–503. IEEE Computer Society, 2006.
[25] F.-S. Tsai and Y.-C. Hsu. STAR: An automatic data path allocator. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2(9):1053–1064, september 1992.
[26] C. Tseng and D. Seiwiorek. Automated synthesis of data paths in digital systems. *IEEE Trans. on Computer-Aided Design*, 1986.
[27] K. Wakabayashi. C-based synthesis experiences with a behavior synthesizer, Cyber. In *DATE '99: Proceedings of the conference on Design, Automation and Test in Europe*, page 83, 1999.